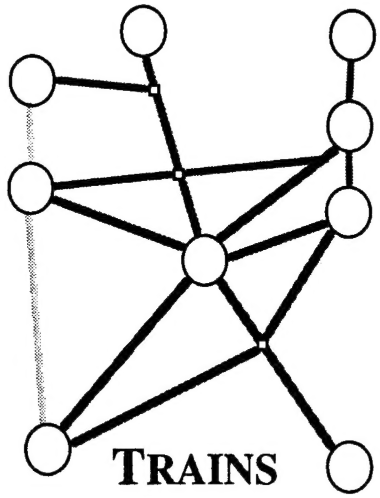

The TRAINS-95 Parsing System: A User's Manual

James F. Allen

TRAINS Technical Note 95-1
September 1995

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The TRAINS-95 Parsing System: A User's Manual[1]

James F. Allen
Dept. of Computer Science
University of Rochester
Rochester, NY 14627
james@cs.rochester.edu

Abstract

This report is a user's manual for the TRAINS-95 parsing system. An accompanying report describes the grammar used in TRAINS-95, and the robust speech act interpretation system, which takes the chart and produces a series of speech acts that best characterize it.

The parser is based on the bottom-up parser described in *Natural Language Understanding, Second Ed.* (Allen, 1994, Chapters 3, 4, and 5). It uses the same formats for the grammar and the lexical entries, and the same basic bottom-up algorithm. There are a number of extensions beyond the basic system described in the book, each of which will be discussed in this report, including

- support for parsing word lattices
- best-first parsing using context-free probabilistic rules
- incremental (word by word) parser with backup for corrections
- hierarchical feature values and extended unification options
- a hierarchical lexicon entry format that simplifies defining large lexicons
- procedural attachment to Chart actions

[1] This research was supported in part by ONR/ARPA research grant no. N00014-92-J-1512. Thanks to Mark Core, Aaron Kaplan, and Drew Simchik for work on the parser at various stages of its development.

REPORT DOCUMENTATION PAGE

Form Approved

OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED technical report

4. TITLE AND SUBTITLE The TRAINS-95 Parsing System: A User's Manual	5. FUNDING NUMBERS ONR/ARPA N00014-92-J-1512
6. AUTHOR(S) James F. Allen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES Computer Science Dept. 734 Computer Studies Bldg. University of Rochester Rochester NY 14627-0226	8. PERFORMING ORGANIZATION
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES) Office of Naval Research Information Systems Arlington VA 22217 ARPA 3701 N. Fairfax Drive Arlington VA 22203	10. SPONSORING / MONITORING AGENCY REPORT NUMBER TN 95-1

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT Distribution of this document is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS charting; chart parsing; natural language	15. NUMBER OF PAGES 15 pages
	16. PRICE CODE free to sponsors; else $2.00

17. SECURITY CLASSIFICATION OF REPORT unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT unclassified	20. LIMITATION OF ABSTRACT UL

Table of Contents

1 The Format of Constituents and Grammatical Rules

In all input and output interactions, constituents are represented as a list starting with the syntactic category of the constituent, followed by an arbitrary number of feature value pairs. For example, here is an NP constituent with an AGR feature 3s, and a SEM feature CONTAINER:

```
(NP (AGR 3s) (SEM CONTAINER))
```

Feature values may be LISP atoms[2], embedded constituents, variables, or constrained variables. A variable can be written in several different forms:

?<atomic name> - an unconstrained variable (e.g., ?A), which will match any value;
(? <atomic name>) - syntactic variant of a simple variable (e.g., ?A);
(? <atomic name> <val1> ... <valn>) - a variable constrained to be one of the indicated values (e.g., (? A 3s 3p));
?!<atomic name> - a variable that matches anything but the empty value "–" (e.g., ?!A).

A special construction allows constituents to be values, and is of the form

```
(% <cat> (<feat> <val>)*).
```

For example, here is a VP constituent with an NP constituent with AGR value 3s as the value of the SUBJ feature:

```
(VP (AGR 3s) (SUBJ (% NP (AGR 3s)))).
```

Note if you omitted the % in this example, then the value of the SUBJ feature would be interpreted simply as a list structure and not as an embedded constituent.

The basic syntax for a grammatical rule is

```
(<lhs constit> <rule id> <rhs constit 1> ... <rhs constit n>)
```

Constituents on the right hand side of a rule may be designated as head constituents by enclosing them in a list with the first element being the atom *head*. The treatment of head features is discussed in section 2.

For example, the following is a rule for a sentence-level constituent S:

```
((s (agr ?a))
  -1>
    (np (agr ?a))
    (head (vp (agr ?a)))
```

This rule has as its left hand side an S constituent with the AGR feature being the variable ?a, a rule identifier of -1>, and two constituents on the right hand side: an NP and a VP, both with an AGR feature that must unify with the AGR feature in the S constituent. The VP is the head constituent.

Rules may also have weights associated with them, and the parser uses a best-first search strategy to build constituents using the rules with the highest weights first. This allows the user to define

[2] In fact, any LISP data structure can be used as a value. The only operation used by the parser on values, however, is EQ. Thus everything looks like an atom to the parser.

context-free probabilistic grammars as described in *Natural Language Understanding* (Allen, 1994). Weights do not have to follow the laws of probability, however. The user may specify any weights they wish on rules in order to affect the search. The weight for a rule is indicated immediately following the rule ID. If not specified, the weight of a rule is set to a default value, which is initially set to 1.0. This can be changed by the user as described in Section 3. Here's rule -1-> again, this time specified to have a weight of .5.

```
((s (agr ?a))
  -1>  .5
   (np (agr ?a))
   (head (vp (agr ?a)))
```

Rules may contain a variable as a constituent on the right hand side. This is useful for allowing subcategorization information in the feature system. For example, here is a rule that will work for any verb that takes a single constituent complement (assuming the lexicon is defined appropriately)

```
((vp)
-vp1>
    (head (v (subcat ?c)))
    ?c)
```

With such a rule, simple transitive verbs would have to have the SUBCAT value (% NP) in the lexicon.

2 Defining Grammars

A grammar is a list of rules, together with a prefix that indicates what input format is being used. The rules above are in what is called "cat format", where the CAT feature is not explicitly present. For example, here is a small grammar in CAT format.

```
(setq *testGrammar1*
 '(cat
  ((s (agr ?a)) -1-> (np (agr ?a)) (vp (agr ?a)))
  ((np (agr ?a)) -2-> (art (agr ?a)) (n (agr ?a)))
  ((vp (agr ?a) (vform ?v)) -3-> (v (agr ?a) (vform ?v) (subcat _none)))
  ((vp (agr ?a) (vform ?v)) -4->
      (v (agr ?a) (vform ?v) (subcat _np)) (np))))
```

Using Head Features

A head feature captures a constraint between the values of the mother constituent to its head subconstituents (see Allen, 1994). Head features can be defined for a set of rules, and are very useful for abbreviating rules with large numbers of systematically defined features. If head feature format is used in a grammar, every rule should have at least one head subconstituent on the left hand side. In this parser, head features are local to specific categories rather than being global to the grammar (say, as in GPSG). Since the overall grammar is defined incrementally from a set of smaller grammars, the head features are not global to the final grammar. Rather, they are local to each cluster of rules defined. The head features for a grammar are specified using the form

(Head features (<cat$_{1}$> <feat$_{1.1}$> ... <feat$_{1,m}$>) ...(<cat$_{n}$> <feat$_{n,1}$> ... <feat$_{n,k}$>)).

For example, here is the same grammar as `*testGrammar1*` above, but in head feature format

```
(setq *testGrammar2*
   '((head features (vp vform agr) (np agr))
     ((s (agr ?a)) -1-> (np (agr ?a)) (head (vp (agr ?a))))
     ((np) -2-> (art (agr ?a)) (head (n (agr ?a))))
     ((vp) -3-> (head (v (subcat _none))))
     ((vp) -4-> (head (v (subcat _np))) (np))))
```

These two grammars would generate exactly the same grammar in the internal format.

Foot Features

A foot feature moves feature values from any subconstituent to the mother (see Allen, 1994). Foot features can only be defined globally at the present time, so using them adds significant overhead to a large grammar. Unlike head features, which are a notational convenience, the behavior of foot features cannot easily be captured in the standard grammar format. When a feature F is defined as a foot feature, then whenever a rule is used, if one of the subconstituents has a value for F, then that value is passed on to the mother constituent. If two subconstituents define a value for feature F, then the values must unify. Foot features are needed to define the behavior of the feature WH, which indicates if a constituent contains a wh-term.

(define-foot-feature <feature name>)
This defines the feature name as a foot feature globally throughout the grammar. For example, the WH feature is defined as a foot feature by (define-foot-feature 'WH).

Declaring Lexical Categories

For improved tracing and the handling of gaps, you need to declare all lexical categories. If you don't use the gap feature, declaring the lexical categories is not essential. The following categories are preset by the system:

n	noun	p	preposition	pp-wrd	words that function like pp phrases
v	verb	aux	auxiliary verb		
adj	adjective	pro	pronoun	name	proper name
art	article	qdet	question determiner	to	the word "to"

Additional lexical categories are defined as follows:

(defineLexicalCategories <list of categories>)
This defines the lexical categories to be the list specified.

(addLexicalCat <category name>)
This adds the indicated lexical categories to the current list of lexical categories.

Lexical Entries

A lexicon consists of a list of word entries of form

```
(<word> <constit>)
```

where the constit is in abbreviated format as described above. Here's a sample lexicon

```
(setq *Lexicon1*
 '((dog (n (agr 3s) (root dog)))
   (dogs (n (agr 3p) (root dog)))
   (pizza (n (agr 3s) (root pizza)))
   (saw (v (agr ?a1) (vform past) (subcat _np) (root see)))
   (barks (v (agr 3s) (vform pres) (subcat _none) (root bark)))
   (the (art (agr 3s) (root the)))))
```

3 Loading and Accessing the Grammar and Lexicon

The following functions are provided to define which grammars and lexicons are to be used by the parser. These functions convert the input formats into the internal formats. For defining the grammar, there are two functions. One redefines the active grammar to a new grammar while the other adds additional rules to the active grammar.

(make-grammar <grammar spec>)
This defines the active grammar to the specified grammar.

(augment-grammar <grammar spec>)
This adds the rules in the specified grammar to the active grammar.

There are two functions for accessing the active grammar:

(get-grammar)
This returns the complete active grammar

(show-grammar <optional rule id> ... <optional-rule-id>)
This prints the rules with the indicated rule IDs. For example,

```
(show-grammar '-s1-> '-vp->)
```

would print out all rules with the ID `-s1->` or `-vp1>`. If no IDs are specified, the entire grammar is printed out.

For the lexicon, a similar pair of functions is provided.

(make-lexicon <list of lexical entries>)
This defines the active lexicon to be the specified lexical entries.

(augment-lexicon <list of lexical entries>)
This adds the lexical entries to the active lexicon

There are several functions for accessing the active lexicon

(worddef <word>)
This returns all the lexical entries for the indicated word, e.g., `(worddef 'to)` returns all the entries for TO.

(get-lexicon)
This returns the complete active lexicon

(get-defined-words)
This returns a list of all the words that are defined in the lexicon.

Setting the Default Rule Probability

If you want to change the default rule probability from 1.0, you may do so using the following function. Note that this default is used when the rules are added to the active grammar. So different defaults could be used for different sections of the grammar if they are added separately.

> **(Set-Default-Rule-Probability <number>)**
> Sets the default rule probability to the indicated number. While meaningful probabilities must fall between 0 and 1, the parser actually accepts any numbers the user wishes to define as it does not depend on the laws of probability in any of its calculations.

4 Running the Online Parser

The parser is called using two functions, each taking a list of atoms (i.e., the words) as its argument, e.g.,

> **(start-BU-parse <list of words>)**
> This initiates a new parse starting with the indicated words (e.g.,

```
   (start-BU-parser '(the dog)))
```

> **(continue-BU-parse <list of words>)**
> This continues the parse with the indicated words(e.g., calling

```
   (continue-BU-parse'(ate the pizza))
```

> after

```
   (start-BU-parser '(the dog)))
```

> will produce the chart for the sentence "the dog ate the pizza".

A useful facility is provided that takes string input and expands contractions and punctuation:

> **(Tokenize <sentence as a string>)**
> This expands contractions and converts punctuation in a string into a list of atoms for the parser. For example, the string "I don't know Avon's size." returns the list (I DO NOT KNOW AVON ^S SIZE PUNC-PERIOD). Table 1 gives example of the expansions that Tokenize handles.

Punctuation is mapped in special symbols: PUNC-PERIOD ("."), PUNC-COLON (":"), PUNC-SEMICOLON (";"), PUNC-COMMA (","), PUNC-QUESTION-MARK ("?"), PUNC-EXCLAMATION-MARK ("!").

Lattice-Based Parsing

The parser also supports a generalized input format that allows it to parse word lattices and other structures. When calling the parser, you may use the following format in place of a single atom presenting a word:

> **(<word atom> <optional modifiers>)**

where the optional modifiers are indicated in keyword format

Abbreviation	Example	Result
Regular n't forms	don't	DO NOT
Regular 'll forms	you'll	YOU WILL
Regular 'd forms	I'd	I WOULD
Regular 're forms	you're	YOU ARE
Regular 've forms	we've	WE HAVE
Possessives 's	George's	GEORGE ^S
Possessives '	engines'	ENGINE ^
Significant blank	I_want	I WANT
o'clock	o'clock	OCLOCK
Contraction	I'm	I AM
Contraction	Let's	LET US

Table 1: Summary of expansions performed by the function Tokenize

:start <start position> - indicates the starting position of the word. Default is the next position in the input

:end <end position> - indicates the ending position of the word. Defaults to the start position plus one

:prob <number> - indicates the "probabilistic" weight of the word. Defaults to 1.0

:filter <constit spec> - indicates the syntactic category and necessary features of the word entry. Only those lexicon entries that unify with this filter are used. If not specified, all entries in the lexicon are used. If the constit spec is an atom, it specifies the syntactic category required. If the constit-spec is a list, it must be of the form (<cat> (<feat> <val>)*), as in (NAME (SEM CITY)).

For example, given the input item `(ate :start 3 :prob .5)`, the parser would look up all lexical entries for ATE and create entries from position 3 to 4 for each with probability .5. Given the input item `(run :start 4 :filter NOUN)`, the parser would only use lexical entries for RUN that are nouns. Given `(Toledo :start 5 :end 8 :filter (NAME (SEM CITY)))`, the parser would only use lexical entries for TOLEDO that are names with SEM feature CITY, and each constituent spans from position 5 to 8.

The input formats may be mixed, and the parser keeps track of the current position throughout, updating this position only when entries are given that do not specify a start position and the default start position is used. For instance, consider the input sequence

```
(GO TO TO LEAVE OH (TOLEDO :start 3 :end 6) AND WAIT)
```

This would create entries for GO from 1 to 2, TO from 2 to 3 and from 3 to 4, LEAVE from 4 to 5, OH from 5 to 6, TOLEDO from 3 to 6, AND from 6 to 7 and WAIT from 7 to 8. Exactly the same interpretation would be given to the input sequence

```
(GO TO (TOLEDO :start 3 :end 6) TO LEAVE OH AND WAIT).
```

Backing Up

Backing up the parser is simple.

(backup <number>)
This causes the parser to back up the indicated number of words. The resulting state is exactly the same as the parser was in before the words that were backed over were ever entered. Warnings: if you use string input with tokenization, remember that contractions are expanded into multiple words, all of which must be counted. If you want to back over the input word "can't", this will require backing over two words: CAN and NOT. Also, if you have words that span multiple positions, they will be deleted if any part of them is backed over.

Setting the Maximum Chart Size

The function ContinueUtterance can be called as long as you wish, until the chart exceeds its maximum size. The max chart size can be found by calling (GetMaxChartSize), and the default size is 40 constituents. You may change this by calling SetMaxChartSize, e.g., (SetMaxChartSize 50).

5 Tracing and Other Control Options

Global Tracing

There are two global levels of tracing provided:

(traceon)
This causes a trace message as each constituent is added to the chart

(verboseon)
This causes a trace message of each arc extension as well.

(verboseoff)
This stops the extended tracing

(traceoff)
This disables all tracing

Specific Tracing

You can trace all activity of a single rule in the grammar, which causes a message to be printed whenever it is extended and when it is completed and adds a constituent to the chart.

(trace-rule <rule-id> ... <opt-rule-id>)
This traces all rules that have the indicated identifier (e.g., `(trace-rule '-1->)` causes a trace message whenever rule `-1->` is extended or is completed). You may trace as many rules simultaneously as you want. To disable individual tracing, you have to use (traceoff).

The default trace message prints all features in every constituent. You can change this by specifying which features you would like to see.

(set-trace-features <list of feature names>)
This will cause all trace messages to only print the indicated features. If the argument is T, then all features are printed. If it is NIL, then no features are printed.

Setting a Break Point

You can set a single break point, where the parser will stop when it adds a constituent that matches an indicated pattern to the chart.

(break-on <constit>)
This sets the parser to stop when a constituent is found that unifies with the constituent specified. For example,

```
(break-on (% S (STYPE (? x DECL YNQ)))
```

would cause the parse to stop when it finds an S constituent with the STYPE feature being DECL or YNQ. The parser retains all its state, so you can continue a parse after stopping it simply by calling `(Continue-BU-parse nil)`. To turn off the break point, simply call break-on with nil as an argument.

6 Accessing the Chart

There are several ways to access the chart built from the last parse:

(show-chart <opt-feat-list> <opt-start-index> <opt-end-index>)
This prints out the complete chart, showing all chart entries as they were built bottom-up. If the <optional feature list> is specified, only the features mentioned are printed. If it is the value T, all features are printed. If the <optional start index> and <optional end index> are specified, then only the constituents between these positions are printed. If the starting position is specified and the end position omitted, all constituents after the starting position are printed. Consider the examples:

(show-chart '(LEX LF)) - display entire chart showing only LEX and LF features
(show-chart '(LEX LF) 3) - display entire chart from position 3 to end showing only LEX and LF features
(show-chart t 3 6) - show chart between positions 3 and 6, showing all features.

(show-best <opt-feat-list> <opt-start-index> <opt-end-index>)
This prints out only the chart entries that span the complete input between the specified indices, or the entire sentence if the indices are not specified. If no constituent spans the entire utterance, this prints the smallest sequence of constituents that cover the entire sentence. See discussion of show-chart above for interpretation of defaults for parameters.

(show-answers <opt-feature-list> <opt-start-index> <opt-end-index>)
This prints out the complete parse tree rooted at each of the constituents found in show-best. This function binds all the variables in the subconstituents as it prints giving a better view of the overall parse tree than the full chart where the constituents are shown as they were defined bottom-up.

(show-constit <constituent-name> <opt-feat-list>)
This prints out the complete parse tree rooted at the indicated constituent. If the optional feature list is specified, then only those features are printed.

7 Handling Gaps

The system supports a basic facility for automatically propagating GAP features and inserting gaps at appropriate locations during the parse. There are functions to enable and disable gap processing. Note that gap processing must be enabled before a grammar is loaded so that the GAP feature is inserted in the rules.

(enableGaps)
This enables gap processing.

(disableGaps)
This disables gap processing.

When gap processing is enabled, the GAP feature is added to rules when they are defined, as described in *Natural Language Understanding, 2nd ed.* Basically, gaps propagate from mother constituent to their head constituent when they are non-lexical, and to the other subconstituents when the head constituent is a lexical constituent.

Since this is a bottom-up parser, some effort has to be made to restrict the arbitrary bottom-up insertion of gaps. To help in this, the user must declare what constituents can participate in gaps. For each constituent type that you wish to participate as gaps, you must call the following function:

(declare-gap-cat <category name> <list of features>)
This declares that the category will be used in GAPs, and the indicate list of features are all the features that will be relevant for unification with these gaps. This defaults just to NP constituents with features AGR, CASE and SEM.

(reset-gap-cats)
Removes all previous declarations concerning categories that can participate in gaps.

Rules that introduce gaps specify the constituent needed using an embedded constituent as the value of the GAP feature. For example, here is a rule for WH questions, which consist of an NP with the WH feature Q, followed by an inverted S with the GAP feature value `(% np (agr ?a))`. Note that the AGR feature of the gap must agree with the AGR feature of the WH NP.

```
((s)
 -5-8-3>
    (np (wh q) (gap -) (agr ?a))
    (head (s (inv +) (gap (% np (agr  ?a))))))
```

8 Debugging Grammars

One nice feature of bottom up algorithms is that they allow for some effective debugging strategies. If you find that a sequence of words doesn't parse how you think it should, try each of the constituents one at a time and see what the parser produces. If you don't get the right analysis, then try additional subparts of the sentence until you find the answer. For example, say you try

```
(start-bu-parse '(the angry man ate the pizza))
```

and it doesn't produce a complete interpretation. You might then try

```
(start-bu-parse '(the angry man))
```

and see if this produces an appropriate NP analysis. Say it does. Then try

```
(start-bu-parse '(ate the pizza))
```

ands see if this produces the appropriate VP analysis. If it doesn't, then there's probably a problem with your VP -> V NP rule, or your lexical entry for "ate". If it does produce the right interpretation, then the problem must be with your S -> NP VP rule (e.g., maybe the rule is missing, or maybe a feature equation is wrong, etc.).

You can also inspect the chart after each word is entered by simply calling the parser incrementally one word at a time and then using the standard chart access functions.

9 Hierarchical Features

The parser supports hierarchical features, where the values are defined in a hierarchy, and two values unify if one is a specialization of the other. Features that use hierarchical values must be declared.

(declare-hierarchical-features <list of features>).
This declares the indicated features as having hierarchical values. For instance,

```
declare-hierarchical-features '(SEM AGR))
```

would declare that only SEM and AGR features take hierarchical values.

The unifier is generalized for hierarchical features so that two values match if one is a specialization of the other according to a pre-defined type hierarchy. This hierarchy is a tree that is declared in a simple list format. Before you can define an actual hierarchy, you must initialize the hierarchy structure:

(init-type-hierarchy)
This initializes the type hierarchy, clearing it if it was previously defined.

(compile-hierarchy <tree in lisp form>)
This extends the existing type hierarchy (or empty hierarchy) with the hierarchical relationships specified as the argument.
For example, the call

value needed for rule	value in chart	result
FIXED-OBJ	CITY	success, since a CITY is a FIXED-OBJ
CITY	FIXED-OBJ	fail, since a FIXED-OBJ is not necessarily a CITY
(? d FIXED-OBJ)	CITY	success, ?d bound to CITY
(? d FIXED-OBJ)	(? e CITY)	success, ?d bound to (? e CITY)
(? d CITY)	(? e FIXED-OBJ)	fail
(? d CITY COMMODITY)	CONTAINER	success, ?d bound to (? f COMMODITY)
(? d CITY COMMODITY)	(? e CITY CONTAINER)	success, ?d and ?e bound to (? f CITY COMMODITY)

Table 2: Examples of feature unification with hierarchical values

```
(compile-hierarchy
 '(PHYS-OBJ
        (FIXED-OBJ
            (CITY)
            (LAKE)
            (COUNTRY)
            (RIVER))
        (MOVABLE-OBJ
            (COMMODITY
            (LIQUID-COMMODITY)
            (SOLID-COMMODITY)
            (CONTAINER))
    ))
```

would define the hierarchy as indicated by the list structure.

Note that in matching hierarchical features, the matching process is not symmetric between the constituent being sought for a rule, and the constituent being matched against in the chart. If a rule needs a constituent with a SEM feature of FIXED-OBJ, then it can unify with an existing constituent with a SEM feature that is a specialization of FIXED-OBJ, such as CITY. But if a rule needs a constituent with a SEM feature of CITY, it will not unify with an existing constituent with SEM FIXED-OBJ, because not all FIXED-OBJs are necessarily CITYs. This is exactly parallel to the use of type hierarchies in knowledge representation systems, where the constituent being sought for the rule is the goal, and the chart is the knowledge base. Given the hierarchy above, Table 2 shows some results of unifying various values. Note that in the last two examples, a new variable is created as the substitution as it draws information from both the value needed and the value in the chart.

10 Enhanced Lexicon Input Capabilities

The parser provides a facility that allows lexicons to be specified in hierarchies with inheritance of features, and which can automatically generate lexical variants (such as plural forms of nouns and the regular verb forms). It also inserts default values for the features LEX, LF and VAR, and other features related to morphological variants of verbs, namely AGR, and VFORM. A function expand is provided that takes a hierarchical specification and generates the standard input format for each entry.

Defining a Hierarchical Lexicon

A hierarchical lexicon is a tree of partial lexical entries, where each leaf node inherits feature values from its ancestor nodes. When multiple values for a feature are defined in the ancestors, the value declared closest to the leaf node (or at the leaf node) is used. The tree is specified as a list structure consisting of nodes and leaf nodes. A node is of the form

```
(:node <list of feature specs> <list of subnodes>)
```

and a leaf node is specified as

```
(:leaf <lexical item> <list of feature specs>).
```

The following default values are added for features if they are not specified:

LEX - set to the word
LF - set to the base form of the word
VAR - set to a new variable

Additional entries are created based on the MORPH feature as follows:

-S-3p - create a plural form by adding an "s", set AGR to 3p
-vb - create the different verb forms using regular morphology, i.e.,
add "s" for AGR 3s
add "ing" for VFORM ING
add "ed" for VFORM PAST and PASTPART

To invoke the enhanced lexical entry facility, use the following function:

(expand <category> <node>)
This creates a list of lexical entries of the indicated category as defined by the tree rooted at the specified node. This includes generating morphological variants as defined by the MORPH feature. The entries can be added to the lexicon by mapping make-lex over the result.

For example, here is a small tree defining a few lexical entries.

```
(expand
 '(:node
    ((AGR 3s) (MORPH -S-3P) (ARGSEM PHYS-OBJ))  ;; the feature values
    ((:node                          ;; first subnode
        ((SEM WEIGHT))
        ((:leaf ton)
         (:leaf pound)
            ))
    (:leaf gallon (SEM VOLUME) (ARGSEM LIQUID))  ;; second subnode
    )))
```

This returns the following lexical entries

Root	Suffix	Action	Example
ends in “Cy”	“s”	replace “y” with “ie”	cities
ends in “Cy”	“ed”	replace “y” with “i”	carried
ends in “e”	begins with “e”	drop final “e”	cared
ends in “h”, “s” or “x”	“s”	insert “e”	finishes
ends in “Vb”, “Vg” or “Vp”	begins with “i” or “e”	double final letter	bagged

Table 3: Simple morphological variants

```
((ton (LEX ton) (LF ton) (VAR V11) (AGR 3s) (MORPH -S-3p) (ARGSEM PHYS-OBJ) (SEM
    WEIGHT))
 (tons (LEX tons) (LF ton) (VAR V12) (AGR 3p) (MORPH -S-3p) (ARGSEM PHYS-OBJ) (SEM
    WEIGHT))
 (pound (LEX pound) (LF pound) (VAR V13) (AGR 3s) (MORPH -S-3p) (ARGSEM PHYS-OBJ) (SEM
    WEIGHT))
 (pounds (LEX pounds) (LF pound) (VAR V14) (AGR 3p) (MORPH -S-3p) (ARGSEM PHYS-OBJ) (SEM
    WEIGHT))
 (gallon (LEX gallon) (LF gallon) (VAR 15) (AGR 3s) (MORPH -S-3p) (ARGSEM LIQUID) (SEM
    VOLUME))
 (gallons (LEX gallons) (LF gallon) (VAR 16) (AGR 3p) (MORPH -S-3p) (ARGSEM LIQUID) (SEM
    VOLUME)))
```

Morphological Variants

The morphological expansion handles simple variants as shown in table 3, where C is any consonant and V any vowel. Note that these are not hard and fast rules. The automatic derivations are a matter of convenience only. Words that do not follow the general rules must be marked as exceptions.

Morphological derivations are triggered by the MORPH features. So one way to deal with exceptions is to omit the relevant MORPH feature and define the entries by hand. For example, the entry for the word *plenty*, which has no plural form, would not have the MORPH feature -S-3P and thus the variant “plenties” would not be constructed. Similarly, irregular verbs could be defined by not adding the -vb MORPH feature and defining each entry by hand. There are enough irregular verb forms, however, that this would be quite cumbersome, so an additional mechanism is provided. A table of verb exceptions is defined that is checked for any entry with the MORPH feature -VB. This table may list any irregular forms of the verb. The format of an entry in this table is

(<root form> <vb-feature>$_1$ <form>$_1$ ... <vb-feature>$_n$ <form>$_n$).

For example, the entry

```
(bring :past brought)
```

indicates that bring has an irregular past form "brought", but the 3s form (brings) and the ing form (bringing) are formed regularly. Unless explicitly indicated, the pastpart form is always taken to be identical to the past form, so the pastpart form here is "brought". An example where the past and pastpart differ is "come"

```
(come :past came :pastpart come).
```

To define exceptions, you call the function

(init-verb-exception-table <list of entries>)
This function defines the set of verbs which have at least one form that is not derivable by the standard methods, e.g.,

```
(init-verb-exception-table
 '(  (come :past came :pastpart come)
        (bring :past brought)))
```

would define entries for "come" and "bring".

11 Procedural Attachment

This parser also allows you to attach arbitrary LISP procedures to the parser. You declare constituent patterns using the ANNOUNCE function, and whenever a constituent is about to be added that matches the pattern, your function is called first. You may then modify the constituent and return it back to the parser. The constituent patterns use the same format as in specifying grammars. Here is an example function that traps all complete NPs, prints them out, and adds a new binary feature, +SEEN.

```
(announce '(np (gap -)) #'testfn)
(defun testfn (entry)
  (Format t "~% Found the NP: ~s~%" entry)
  (setFvalue entry 'SEEN '+)
  ;; must return the entry if it is to be added to the chart
  entry)
```

Note that if you subsequently edit the definition of TEST, you may need to clear the old announcements and make a new one. The function INIT-ATTACHMENTS removes all previous announced patterns, e.g.,

```
(init-attachments)
(announce '(np (gap -)) #'testfn)
```

In general, you will have to dig into the code if you want to modify parts of a constituent. But here are a few of the more common functions.

(GetFvalue <entry> <featurename>)
This returns a feature value, e.g., (getFvalue entry 'SEEN) would return the value of the feature SEEN.

(SetFvalue <entry> <featurename> <value>)
This sets a feature value, e.g., (setFvalue entry 'SEEN '+) sets the feature value SEEN to +

(SetProbValue <entry> <number>)
This sets the probability of the entry, e.g., (setProbValue entry .5) sets the probability to .5.

Note that if you reduce the probability of a constituent so that it is not now the highest ranked constituent on the agenda, it will be placed back onto the agenda to be added to the chart later. This means you will see the same constituent again if it returns to the top of the stack.

If you want to stop the parser from within an attached function, call the function:

> **(suspendParse)**
> This will cause the parser to stop after it processes the current entry. In other words, if your function returns the entry, it is added to the chart or back to the agenda as normal, and then the parser stops. The parser can be resumed simply by calling continue-BU-parse with nil as an argument since the agenda is not cleared.

12 Obtaining the System

The parser is available via anonymous ftp from ftp.cs.rochester.edu in the directory pub/u/james/parser-95.

13 References

Allen, J.F. *Natural Language Understanding, Second Edition,* Benjamin-Cummings Pub Co., 1994.